Linear Algebra and Scientific Python

Dr. Xiaowei Huang

https://cgi.csc.liv.ac.uk/~xiaowei/

Topics

- Linear algebra
  - Scalars, vectors, matrices, tensors
  - Multiplying matrices/vectors
- Introduction to Scientific Python
  - Numpy
  - Scipy
  - Matplotlib

Linear Algebra For Machine Learning

Scalar

- Single number
- Represented in lower-case italic *x*
    - E.g., let $x \in \mathbb{R}$ be the slope of the line
        - Defining a real-valued scalar
    - E.g., let $n \in \mathbb{N}$ be the number of units
        - Defining a natural number scalar

Vector

- An array of numbers
- Arranged in order
- Each no. identified by an index
- Vectors are shown in lower-case bold
- If each element is in *R* then x is in R^n
- We think of vectors as points in space
  - Each element gives coordinate along an axis

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \Rightarrow \mathbf{x}^T = \left[x_1, x_2, \dots, x_n\right]$$

Matrix

- 2-D array of numbers
- Each element identified by two indices
- Denoted by bold typeface ***A***
- Elements indicated as $A_{m,n}$
  - E.g.,

$$\mathbf{A} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$$

- *A*[i:] is *i*th row of *A*, *A*[:j] is *j*th column of ***A***
- If *A* has shape of height *m* and width *n* with real-values then $\mathbf{A} = \mathbb{R}^{m \times n}$

Tensor

- Sometimes need an array with more than two axes
- An array arranged on a regular grid with variable number of axes is referred to as a tensor
- Denote a tensor with bold typeface: **A**
- Element *(i,j,k)* of tensor denoted by $A_{i,j,k}$

Transpose of a Matrix

- Mirror image across principal diagonal

$$A = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} x_{11} & x_{21} & x_{31} \\ x_{12} & x_{22} & x_{32} \\ x_{13} & x_{23} & x_{33} \end{bmatrix}$$

- Vectors are matrices with a single column
  - Often written in-line using transpose

$$\mathbf{x} = [x_1,..,x_n]^{\mathrm{T}}$$

- Since a scalar is a matrix with one element $a = a^{\mathrm{T}}$

Linear Transformation

$$A\mathbf{x} = \mathbf{b}$$

- where $A \in \mathrm{R}^{n \times n}$ and $\mathbf{b} \in \mathrm{R}^n$

$$A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = b_1$$
$$A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = b_2$$
$$\ldots$$
$$A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n = b_n$$

n equations in *n* unknowns

Linear Transformation

$$A\mathbf{x} = \mathbf{b}$$

- where $A \in \mathrm{R}^{n \times n}$ and $\mathbf{b} \in \mathrm{R}^n$
- More explicitly

$$A = \begin{bmatrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & \vdots & \vdots \\ A_{n,1} & \cdots & A_{nn} \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

$n \times n$ $\quad$ $n \times 1$ $\quad$ $n \times 1$

Can view *A* as a *linear transformation* of vector ***x*** to vector ***b***

- Sometimes we wish to solve for the unknowns $x = \{x_1,..,x_n\}$ when *A* and ***b*** provide constraints

Identity and Inverse Matrices

- Matrix inversion is a powerful tool to analytically solve $A\boldsymbol{x}=\boldsymbol{b}$
- Needs concept of Identity matrix
- Identity matrix does not change value of vector
- when we multiply the vector by identity matrix
  - Denote identity matrix that preserves n-dimensional vectors as I_n
  - Formally $I_n \in \mathbb{R}^{n\times n}$ and $\forall \mathbf{x} \in \mathbb{R}^n$, $I_n\, \mathbf{x} = \mathbf{x}$
  - Example of I_3

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrix Inverse

- Inverse of square matrix A defined as $A^{-1}A=I_n$
- We can now solve $A\boldsymbol{x}=\boldsymbol{b}$ as follows:

$$A\boldsymbol{x}=\boldsymbol{b}$$

$$A^{-1}A\boldsymbol{x} = A^{-1}\boldsymbol{b}$$

$$I_n\,\boldsymbol{x} = A^{-1}\boldsymbol{b}$$

$$\boldsymbol{x} = A^{-1}\boldsymbol{b}$$

- This depends on being able to find A^{-1}
- If A^{-1} exists there are several methods for finding it

Solving Simultaneous equations

- $A\boldsymbol{x} = \boldsymbol{b}$
- Two closed-form solutions
  - Matrix inversion $\boldsymbol{x}=A^{-1}\boldsymbol{b}$
  - Gaussian elimination

Norms

- Used for measuring the size of a vector
- Norms map vectors to non-negative values
- Norm of vector ***x*** is distance from origin to ***x***
  - It is any function *f* that satisfies:

$$f(\boldsymbol{x}) = 0 \Rightarrow \boldsymbol{x} = 0$$

$$f(\boldsymbol{x}+\boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y}) \quad \text{Triangle Inequality}$$

$$\forall \alpha \in \mathbb{R} \quad f(\alpha \boldsymbol{x}) = |\alpha| f(\boldsymbol{x})$$

L^P Norm

- Definition

$$||x||_p = (\sum_i |x_i|^p)^{\frac{1}{p}}$$

L^P Norm

- Definition $||x||_p = (\sum_i |x_i|^p)^{\frac{1}{p}}$
- L^2 Norm
  - Called Euclidean norm, written simply as $||x||$
  - Squared Euclidean norm is same as $x^T x$

$$||x||_2 = \sqrt{\sum_i |x_i|^2}$$

$$= \sqrt{x^T x}$$

L^P Norm

- Definition $$||x||_p = (\sum_i |x_i|^p)^{\frac{1}{p}}$$
- L^1 Norm
  - also called Manhattan distance

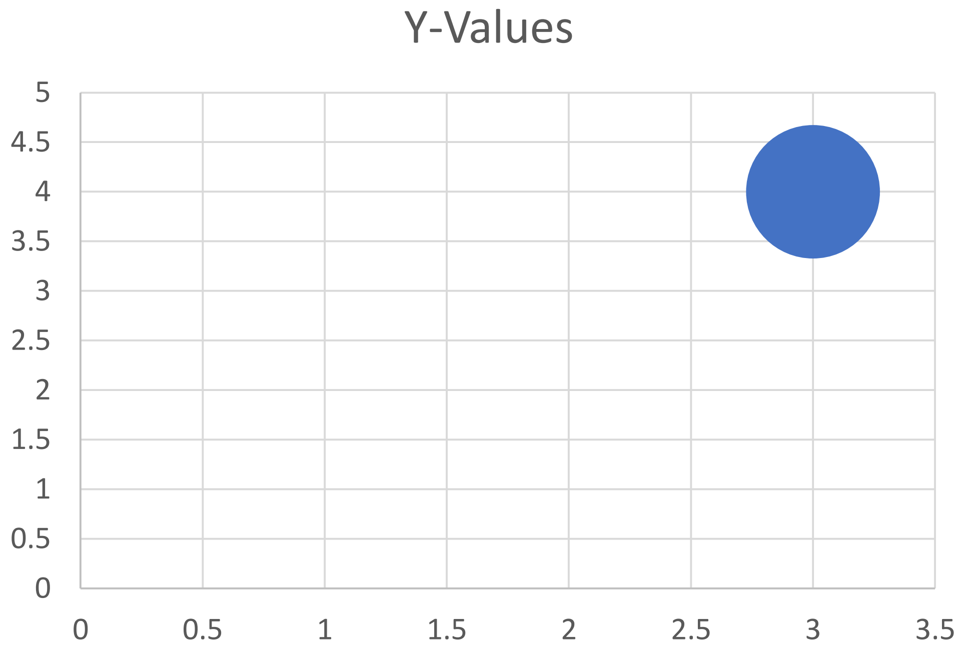
$$||x||_1 \quad = \quad \sum_i |x_i|$$

L^P Norm

- Definition $||x||_p = (\sum_i |x_i|^p)^{\frac{1}{p}}$
- L^∞ Norm
  - also called max norm

$$||x||_\infty \quad = \quad \max_i |x_i|$$

Norms of two-dimensional Point



X = (3,4)

$||x||_1$ = 3+4 = 7

$||x||_2 = \sqrt{3^2 + 4^2} = 5$

$||x||_{\infty} = \max\{3, 4\} = 4$

$$||x||_1 \quad = \quad \sum_i |x_i|$$

$$||x||_2 = \sqrt{\sum_i |x_i|^2}$$

$$||x||_\infty \quad = \quad \max_i |x_i|$$

Size of a Matrix

- Frobenius norm

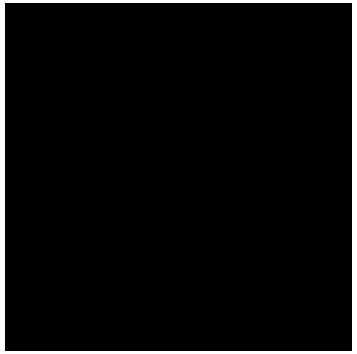
$$||A||_F = \Big(\sum_{i,j} A_{i,j}^2\Big)^{\frac{1}{2}}$$

- It is analogous to L^2 norm of a vector

Image distance

-

=



$$\begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,32} \\ x_{2,1} & x_{22} & \cdots & x_{2,32} \\ \vdots & \vdots & \ddots & \vdots \\ x_{32,1} & x_{32,2} & \cdots & x_{32,32} \end{bmatrix} - \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,32} \\ y_{2,1} & y_{22} & \cdots & y_{2,32} \\ \vdots & \vdots & \ddots & \vdots \\ y_{32,1} & y_{32,2} & \cdots & y_{32,32} \end{bmatrix} = \begin{bmatrix} z_{1,1} & z_{1,2} & \cdots & z_{1,32} \\ z_{2,1} & z_{22} & \cdots & z_{2,32} \\ \vdots & \vdots & \ddots & \vdots \\ z_{32,1} & z_{32,2} & \cdots & z_{32,32} \end{bmatrix}$$

L^1 distance between X and Y: $\sum_{i,j} |z_{i,j}| = \sum_{i,j} |x_{i,j} - y_{i,j}|$

L^2 distance between X and Y: $\sqrt{\sum_{i,j} z_{i,j}^2} = \sqrt{\sum_{i,j} (x_{i,j} - y_{i,j})^2}$

L^∞ distance between X and Y:

$$\max_{i,j} |z_{i,j}| = \max_{i,j} |x_{i,j} - y_{i,j}|$$

Introduction to Scientific Python

Numpy

- Fundamental package for scientific computing with Python
- N-dimensional array object
- Linear algebra, Fourier transform, random number capabilities
- Building block for other packages (e.g. Scipy)
- Open source

import numpy as np

- Basics:

```
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
print A
# [[1 2 3]
#  [4 5 6]]

Af = np.array([1, 2, 3], float)
```

- Slicing as usual

More basics

```
np.arange(0, 1, 0.2)
# array([ 0. ,  0.2,  0.4,  0.6,  0.8])

np.linspace(0, 2*np.pi, 4)
# array([ 0.0,  2.09,  4.18, 6.28])

A = np.zeros((2,3))
# array([[ 0.,  0.,  0.],
#         [ 0.,  0.,  0.]])
# np.ones, np.diag
A.shape
# (2, 3)
```

numpy.arange:
evenly spaced values within a given interval.

numpy.linspace:
evenly spaced numbers over a specified interval.

More basics

```
np.random.random((2,3))
# array([[ 0.78084261,  0.64328818,  0.55380341],
#         [ 0.24611092,  0.37011213,  0.83313416]])

a = np.random.normal(loc=1.0, scale=2.0, size=(2,2))
# array([[ 2.87799514,  0.6284259 ],
#         [ 3.10683164,  2.05324587]])

np.savetxt("a_out.txt", a)
# save to file
b = np.loadtxt("a_out.txt")
# read from file
```

numpy.random.normal:
Draw random samples from a normal (Gaussian) distribution

loc: mean
scale: standard deviation

Arrays are mutable

```
A = np.zeros((2, 2))
# array([[ 0.,  0.],
#        [ 0.,  0.]])
C = A
C[0, 0] = 1

print A
# [[ 1.  0.]
#  [ 0.  0.]]
```

Array attributes

```
a = np.arange(10).reshape((2,5))

a.ndim    # 2 dimension
a.shape   # (2, 5) shape of array
a.size    # 10 # of elements
a.T       # transpose
a.dtype   # data type
```

numpy.reshape:
Gives a new shape to an array without changing its data.

Basic operations

- Arithmetic operators: elementwise application

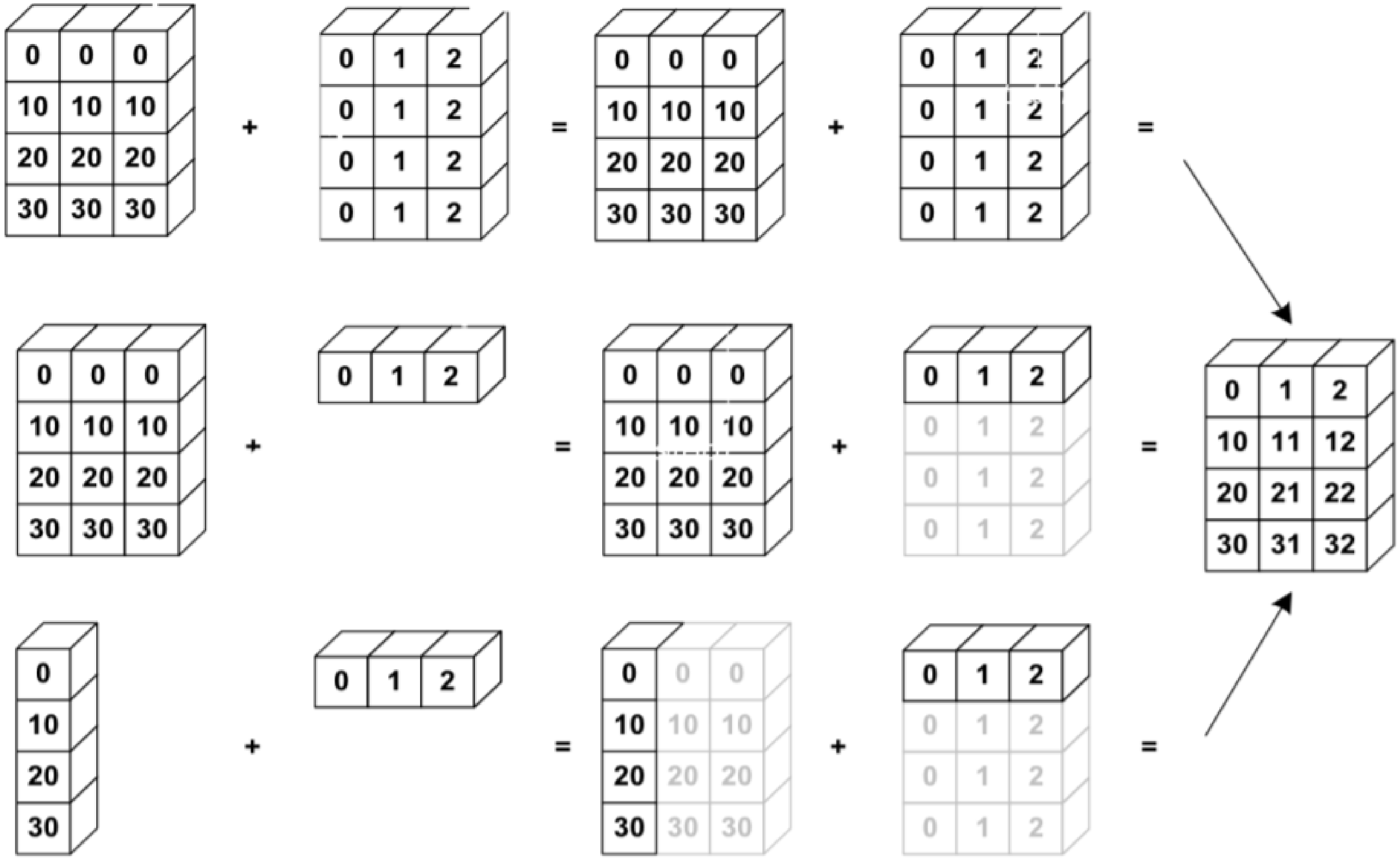
```
a = np.arange(4)
# array([0, 1, 2, 3])

b = np.array([2, 3, 2, 4])

a * b  # array([ 0,  3,  4, 12])
b - a  # array([2, 2, 0, 1])

c = [2, 3, 4, 5]
a * c  # array([ 0,  3,  8, 15])
```

- Also, we can use += and *=.

Array broadcasting

- When operating on two arrays, numpy compares shapes. Two dimensions are compatible when
  - They are of equal size
  - One of them is 1

Array broadcasting



Array broadcasting with scalars

- This also allows us to add a constant to a matrix or multiply a matrix by a constant

```
A = np.ones((3,3))

print 3 * A - 1
# [[ 2.  2.  2.]
#  [ 2.  2.  2.]
#  [ 2.  2.  2.]]
```

Vector operations

- inner product
- outer product
- dot product (matrix multiplication)

```
# note: numpy automatically converts lists
u = [1, 2, 3]
v = [1, 1, 1]

np.inner(u, v)
# 6
np.outer(u, v)
# array([[1, 1, 1],
#        [2, 2, 2],
#        [3, 3, 3]])
np.dot(u, v)
# 6
```

Matrix operations

- First, define some matrices:

```
A = np.ones((3, 2))
# array([[ 1.,  1.],
#        [ 1.,  1.],
#        [ 1.,  1.]])
A.T
# array([[ 1.,  1.,  1.],
#        [ 1.,  1.,  1.]])

B = np.ones((2, 3))
# array([[ 1.,  1.,  1.],
#        [ 1.,  1.,  1.]])
```

Matrix operations

```
np.dot(A, B)
# array([[ 2.,  2.,  2.],
#        [ 2.,  2.,  2.],
#        [ 2.,  2.,  2.]])

np.dot(B, A)
# array([[ 3.,  3.],
#        [ 3.,  3.]])

np.dot(B.T, A.T)
# array([[ 2.,  2.,  2.],
#        [ 2.,  2.,  2.],
#        [ 2.,  2.,  2.]])

np.dot(A, B.T)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# ValueError: shapes (3,2) and (3,2) not aligned:  ...
# ... 2 (dim 1) != 3 (dim 0)
```

np.dot(a,b)

If f both *a* and *b* are 1-D arrays, it is inner product of vectors

If both *a* and *b* are 2-D arrays, it is matrix multiplication

Operations along axes

```
a = np.random.random((2,3))
# array([[ 0.9190687 ,  0.36497813,  0.75644216],
#        [ 0.91938241,  0.08599547,  0.49544003]])
a.sum()
# 3.5413068994445549
a.sum(axis=0)  # column sum
# array([ 1.83845111,  0.4509736 ,  1.25188219])
a.cumsum()
# array([ 0.9190687 ,  1.28404683,  2.04048899,  2.9598714 ,
#         3.04586687,  3.5413069 ])
a.cumsum(axis=1)  # cumulative row sum
# array([[ 0.9190687 ,  1.28404683,  2.04048899],
#        [ 0.91938241,  1.00537788,  1.50081791]])
a.min()
# 0.0859954690403677
a.max(axis=0)
# array([ 0.91938241,  0.36497813,  0.75644216])
```

Slicing arrays

- More advanced slicing

```
a = np.random.random((4,5))

a[2, :]
# third row, all columns
a[1:3]
# 2nd, 3rd row, all columns
a[:, 2:4]
# all rows, columns 3 and 4
```

Iterating over arrays

- Iterating over multidimensional arrays is done with respect to the first axis: **for row in A**
- Looping over all elements: **for element in A.flat**

Reshaping

- Reshape
  - using reshape. Total size must remain the same.

- Resize
  - using resize, always works: chopping or appending zeros
  - First dimension has ‘priority’, so beware of unexpected results

Matrix operations

- import numpy.linalg

`eye(3)`	Identity matrix
`trace(A)`	Trace
`column_stack((A,B))`	Stack column wise
`row_stack((A,B,A))`	Stack row wise

Linear algebra

```
import numpy.linalg
```

`qr`	Computes the QR decomposition
`cholesky`	Computes the Cholesky decomposition
`inv(A)`	Inverse
`solve(A,b)`	Solves $Ax = b$ for A full rank
`lstsq(A,b)`	Solves $\arg\min_x \|Ax - b\|_2$
`eig(A)`	Eigenvalue decomposition
`eig(A)`	Eigenvalue decomposition for symmetric or hermitian
`eigvals(A)`	Computes eigenvalues.
`svd(A, full)`	Singular value decomposition
`pinv(A)`	Computes pseudo-inverse of A

Fourier transform

```
import numpy.fft
```

- `fft` 1-dimensional DFT
- `fft2` 2-dimensional DFT
- `fftn` N-dimensional DFT
- `ifft` 1-dimensional inverse DFT (etc.)
- `rfft` Real DFT (1-dim)
- `ifft` Imaginary DFT (1-dim)

Random sampling

```
import numpy.random
```

`rand(d0,d1,...,dn)`	Random values in a given shape
`randn(d0, d1, ...,dn)`	Random standard normal
`randint(lo, hi, size)`	Random integers [lo, hi)
`choice(a, size, repl, p)`	Sample from a
`shuffle(a)`	Permutation (in-place)
`permutation(a)`	Permutation (new array)

Distributions in random

`import numpy.random`

The list of distributions to sample from is quite long, and includes

- `beta`
- `binomial`
- `chisquare`
- `exponential`
- `dirichlet`
- `gamma`
- `laplace`
- `lognormal`
- `pareto`
- `poisson`
- `power`